Starburst: A Cost-aware Scheduler for Hybrid Cloud

背景

AI和数据分析的需求越来越高，许多企业都构建了自己的大规模、多租户、具备高端GPU的私有云，它们往往采用自定义的调度策略。这些私有云在一些特定事件时（截止日期或产品发布），仍会**面临大量的作业突发，导致高排队时延。**

随着越来越多企业选择了使用混合云架构，而不是一味地投资到硬件。

我们需要将集群调度器转化为云版本（不仅管理本地集群上的调度策略，还管理等待策略），并且需要在**云成本**和**调度目标**（如JCT）之间进行权衡。然而，云的有限资源为这类的调度器带来的新的挑战，它们**不仅需要在云和固定集群间分配任务，还需要感知云的开销。**

一个本地集群**满载时**，若一个作业到达，调度器有两种选择

- 让它等待
    - Constant-Wait
    - No-Short-Jobs：把短作业立即上送给云。
- 立即上交给云端

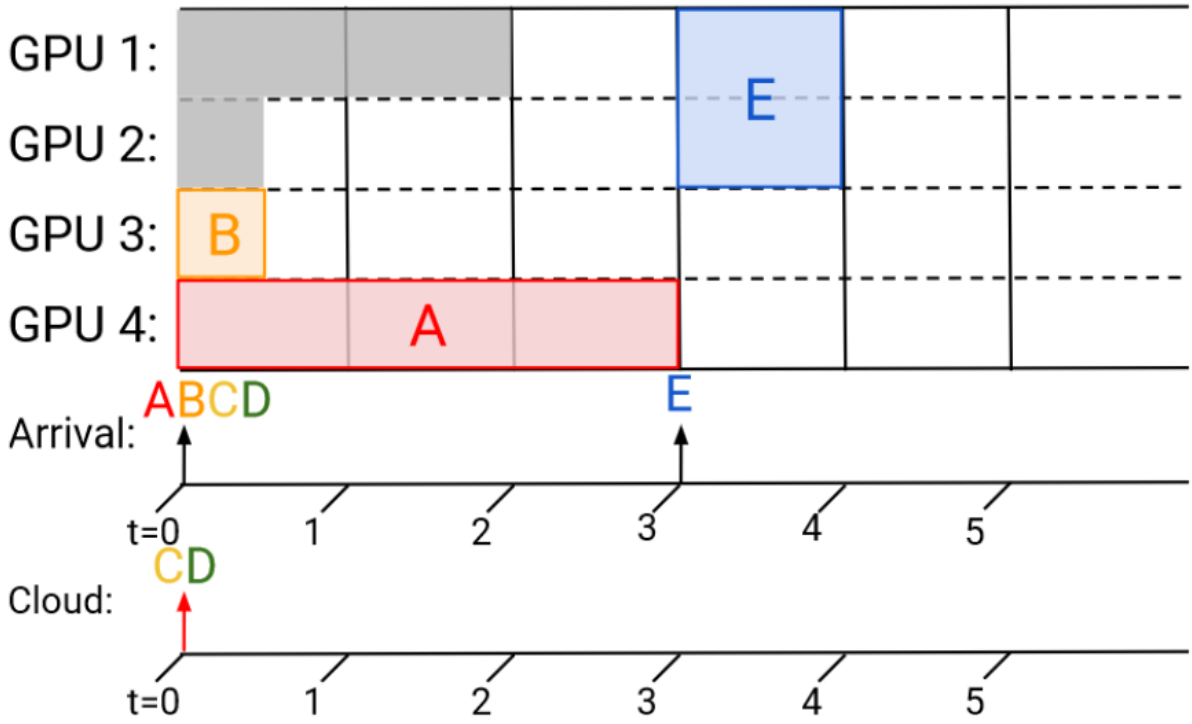
现有的具有云自动伸缩的集群管理器选择了后一种策略，我们称之为No - Wait。消除了排队延迟，降低了平均作业完成时间(JCT)，但由于集群利用率低，引入了高昂的云成本。

cloud-enabled调度器有多个目标，包括云开销，以及传统的JCT，throughput，fairness。**云开销和传统目标是冲突的**，因此存在pareto曲线。

动机

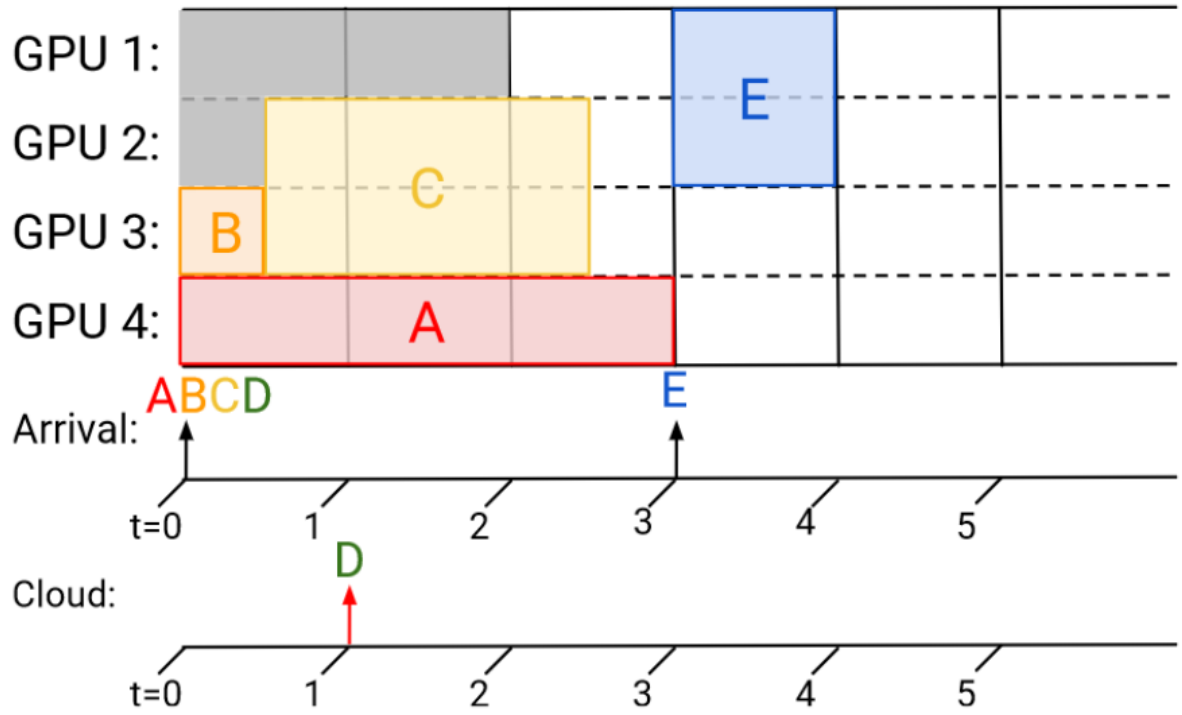
由于现有的cloud-enabled调度器，在处理满载时到达的job，基本都是No - Wait策略。

假如有4个job，分别需要1、1、2、4个GPU资源。



C、D立马被上送到云端，导致t=1时刻后的CPU2,3的资源闲置。

一些工作也提出了constant的等待时间策略。但常数C难以选择。



这里C为1，作业C在t=0.5时，成功执行。但是，作业D是一个更大的4-GPU作业，需要8h执行，没有等到需要的GPU，被上送到了云。

为云带来了高的cost，也降低了本地集群的利用率。

这里可以看出，长的等待时间能够减小云开销，对于长作业也是可以接受的，但是也会阻塞短作业。

等待策略带来的问题：

- **大作业的饥饿。**当作业**被分配的等待时间不足时发生**，大作业基本都会上送到云，不会在本地集群运行，对于本地集群饥饿。这会导致低的集群利用率，因为需要8-GPU以上的大作业占总成本的65 - 99 %，同时，小作业会频繁的在集群中调度，由于它们的高周转性，我们在执行轨迹中观察到了频繁、短暂的资源闲置。
- **队头阻塞。**当作业**被分配较长的等待时间**和**系统负载超过集群容量**(s≥1)时发生，队头的大作业会阻止后面的作业调度，这将导致本地集群的资源闲置。等待时间越长越严重。

关键

与作业计算成本成正比的等待时间能更好的平衡云cost和JCT。

可以更好地利用集群的大型昂贵作业，等待时间足够长以在集群上运行；而规模较小的作业等待时间较短，成本较低，更有可能在云上运行。

调度器必须在分配给本地集群许多小作业和等待足够的资源来调度大作业之间进行平衡，其中分配给本地集群较多的小作业存在将昂贵的大作业发送到云上的风险，而大作业的等待则会阻碍小作业的执行。

贡献

提出了Starburst，可以动态控制作业等待时间。

- 对于大作业，分配**更长**的等待时间，增加它们在**集群**上运行的机会
- 对于小作业，分配**短**的等待时间，增加它们在**云上**运行的机会

同时，对于不同的工作负载和预算限制，也给系统管理员提供了一个简单的**等待预算框架**，通过一个**旋钮**，来提供可配置性。

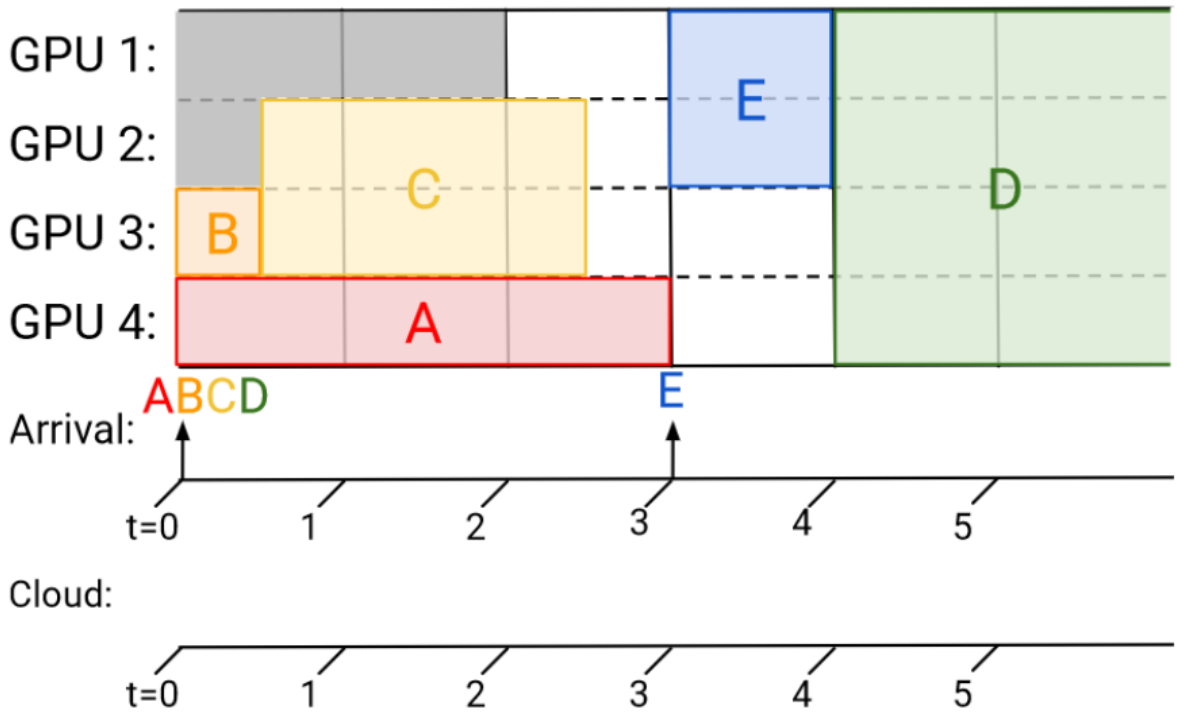
这被配置为一个**P%**，代表管理员愿意接受的平均作业运行时间增加的最大百分比。

根据作业特征，**动态分配等待时间。**

- Compute-Wait：假设知道作业运行的相关知识。根据作业的**预计运行时间与请求资源的乘积**分配。
- Star-Wait：没有先验知识时，运行时和资源感知。

能够从云上抢占执行时间阈值较小的作业回本地，然后分配等待时间。

为了避免长作业的长时间等待对短作业的阻塞，Starburst在队列中大肆**乱序执行作业。**



这里作业E在D之前乱序执行，同时D分配到的等待时间，使得作业D能够在本地集群行运行。

让Starburst的cost-JCT曲线向pareto效率方向移动。

关于Starburst

研究中的前提假设

- 假设提交的作业是运行到完工的批处理作业。
- 假设一个作业被抢占时，它在执行过程中所取得的进展将会丢失。（作者引入了可检查点作业的概念，近似为一系列较小的不可检查点作业）
- 假设作业的运行时间在集群和云环境中都是相似的。（作者忽略了本地集群和云环境中数据局部性和多样化硬件的潜在优势，但简化了模型）

动态等待

Compute Wait

减小大作业对本地集群的饥饿，并让更多的小作业上云。

知道运行时间时，等待时间的计算很简单：

$$w^*(j) = \alpha \cdot \left(\sum_{k=1}^{n} W_k \cdot r_{jk} \right) \cdot t_j$$

对于作业j所需的资源rjk，乘上资源类型k的相对成本，并加起来，乘上t的运行时间tj，和等待预算的超参数α。

作者只考虑了CPU, GPU两种资源，W1:W2=50:1

Without time estimates（Star-Wait）

不需要知道作业运行时间。表现略逊于compute wait。

依赖于一个抢占机制，如果云上的作业运行时间超过Tmax时间，则终止并将其迁移回集群。

$$w^*(j) = \begin{cases} 0 & \text{if } j.preempted = \text{false} \\ \beta \cdot \left(\sum_{k=1}^{n} W_k \cdot r_{jk}\right) & \text{if } j.preempted = \text{true} \end{cases}$$

初始为no-wait策略，本地集群满载时，**直接上送到云**，并执行。

若在云上执行**超过Tmax**，它将被抢占，并重新分配到本地集群。根据作业所需资源，乘上超参数β，分配等待时间。

若还无法在本地运行，则重新上云，并**不再被抢占。**

$$\beta_r = \frac{B}{\sum_j r_j} = \frac{PTN}{(\sum_j r_j) \cdot 100} \approx \frac{PT}{100 \cdot \mathbb{E}[r]}$$

$$\beta = \frac{B}{\sum_j \mathbb{1}_{\text{preempt}} \cdot r_j} \approx \frac{PT}{100 \cdot \mathbb{E}[\mathbb{1}_{\text{preempt}}] \cdot \mathbb{E}[r]} \approx \frac{\beta_r}{p_1 \cdot p_2}$$

T表示作业平均运行时间，N表示作业数，E[r]表示平均请求资源大小，1preempt是一个指示函数，由两个概率p1（最初由no-wait上送到云的作业的比例）、p2（云上超过Tmax的作业的比例）决定。

为了简化β的运算，使用了一种近似，β ≈ X · βr，在实验中的star-wait中，X=3.

乱序调度

基于比例的等待时间计算会带来HOL阻塞（队头阻塞）。

提出了out of order（OO）调度。

简单的循环遍历作业队列，并在集群可用时，贪婪的选择任何可以适合的作业。

等待预算B

定义了所有工作的最大可能的等待时间之和。

B = PTN/100。其中，P为管理员指定的超过平均作业运行时间T的百分比，N为作业数。

- 对于Constant-Wait，B在所有作业上平分。C = B/N = PT/100。

- 对于Compute-Wait，假设作业大小和运行时间之间独立，α仅依赖于P和平均作业大小（所需资源）E[r]。

$$\alpha = \frac{B}{\sum_j r_j t_j} \approx \frac{PTN}{N \cdot \mathbb{E}[r] \cdot T \cdot 100} = \frac{P}{100 \cdot \mathbb{E}[r]}$$

在作者的workloads中，这种独立性是成立的，作业大小和运行时间之间的相关性很小（r<=0.15）。

综上，starbrust只需要知道**工作负载的平均运行时间和大小**，就可以决定等待策略的超参数。

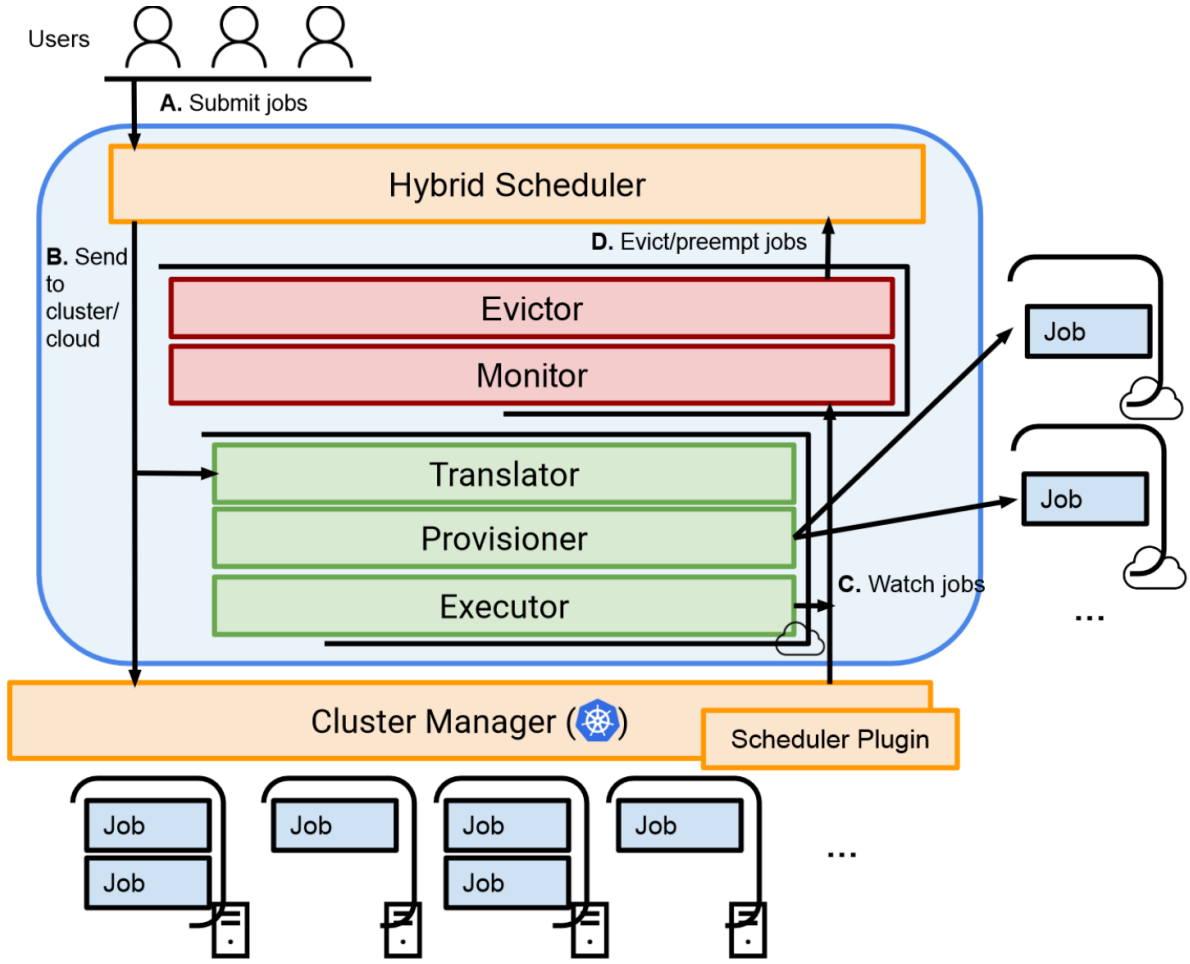
作业的止步和立即上云

一些到达的作业被标记为w∗(j) = −1，可以立刻被卸载到云。

- CPU-only的作业。可以被立刻发送到云
  - GPU价格至少比CPU高一个数量级，将一些gpu作业在本地集群执行，减轻云端的cost
  - 调度CPU作业也会导致资源搁浅，活跃的CPU作业可以阻止GPU作业在云上的调度
- Starburst的作业队列长度上界为Qmax，若超过，作业也会立刻上云。

架构

云集群由来自不同云供应商和地区的独立云实例组成。



- 调度器
  - 混合调度器：控制跨集群的作业调度。调度作业到本地集群或云。**内部集成了排队策略，并执行OO调度。**
  - 集群管理器：**只需要关心装箱策略。**调度作业到节点。实现了自己的调度策略。
- 重调度器：确定了作业的**终止**和**重提交到混合调度器**的准则
  - 监控器：通过集群管理器和云端对作业进行持续观测，实时更新作业状态和作业统计。
  - 驱逐器：通过监控器，检查：1、超过分配时间的作业。2、长期运行的未被抢占的云作业。终止这些作业并重新提交给混合调度器。
- 云编排器：管理云上的作业执行。为每一组云实例假设一个作业，并在作业完成或抢占时终止该实例。
  - 翻译器：用户为集群管理器提供作业规范，翻译器**为执行器翻译**，以便在云上处理和运行
  - 配置器：评估一个作业的资源请求，并提供最合适的云资源，以紧密匹配作业的需求
  - 执行器：执行者负责作业在指定VM上的实际执行

实现

- Simulator实现：一个由Alg.1循环控制的，连续时间离散事件（提交，超时，抢占，完成时触发）激活模型。提供了各种配置参数，如等待策略，等待预算B。
- 现实实现：在本地Kubernetes集群和云集群上实现。
  - 云集群由Skypilot配置，在GCP的不同地区上进行实例化。
  - 混合调度器采用异步事件驱动。
    - JobAddEvent事件：用户提交或Evictor重提交。
    - 使用WaitPolicy类，决定作业等待阈值，并作为K8s注释附加到作业上。
    - 通过内部队列跟踪作业状态，并实现了OO调度。
  - 利用K8s的python API，开发了自定义调度器插件Chakra。使用best-fit binpacking策略，利好批作业。
  - 开发了一个Informer类。维护作业的最新状态，供Evictor使用。
  - 重调度的作业被注释为ToCloud、ToCluster、Preempted。
  - Skypilot与ToCloud的作业交互，通过Translator，将K8s作业的YAML转化为Skypilot的API调用格式。
    - Skypilot管理Provisioner和Executor。
    - 利用Skypilot虚拟机上的Ray提供的作业提交API，管理作业的分发执行。

实验

环境设置

私有集群由4个节点组成，共有32个NVIDIA V100 GPU和384个vCPU

云虚拟机是通过 **Skypilot** 动态配置的，具体是在 **Google Cloud Platform（GCP）** 的不同地区和区上进行实例化

starburst调度器单独在一个GCP实例运行。

Metrics

云开销节省，平均JCT，集群资源利用率

Baselines

作者评估了两种starburst：1、有时间估计的compute-wait 2、没有时间估计的star-wait。

与**No - Wait** (w (j) = 0)、**Constant Wait** (w (j) = C)和最新的**Constant - Wait + No- SJ** (No Short Jobs)对比。

数据集

real-life负载

评估了starburst在**AI微调**作业上的性能。批作业包括CV和NLP。

对于我们的workload trace的生成：

1. 首先通过从泊松过程λ模拟**作业到达时间**，使用指数分布1 / μ模拟**作业运行时间**，生成一个虚拟迹。（目的是生成一系列作业到达的时间点和对应的运行时间）
2. 从一个与 **Microsoft Philly trace** 相匹配的**类别分布（categorical distribution）**中采样，以确定每个作业所需的 GPU 数量。（GPU 数量是基于实际的工作负载数据，反映了在该 trace 中 GPU 资源请求的真实分布）
3. 为虚拟轨迹中的每个作业找到**与其资源请求和运行时间最接近的实际训练作业。**确保生成的虚拟轨迹在资源请求和运行时间上尽可能真实反映实际情况。

模拟器的trace

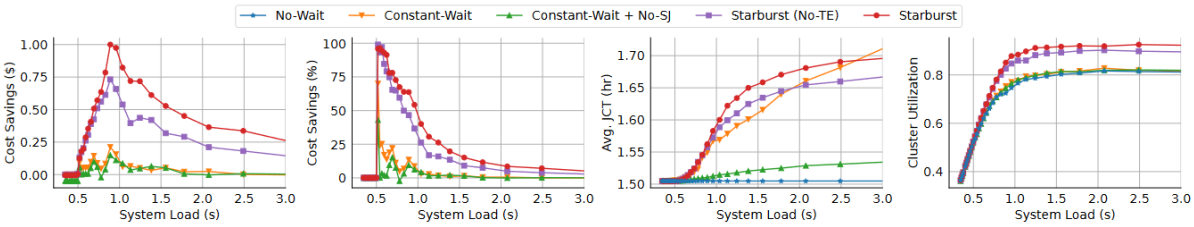
模拟实验依赖于两个公开的生产trace，Philly和Helios。

- 直接在原始trace上评估，时间戳决定了作业何时到达。
- 在合成trace上评估，改变相对于泊松分布λ的到达率来调节系统负载。在一个具有512个GPU的模拟64节点集群上进行了评估。

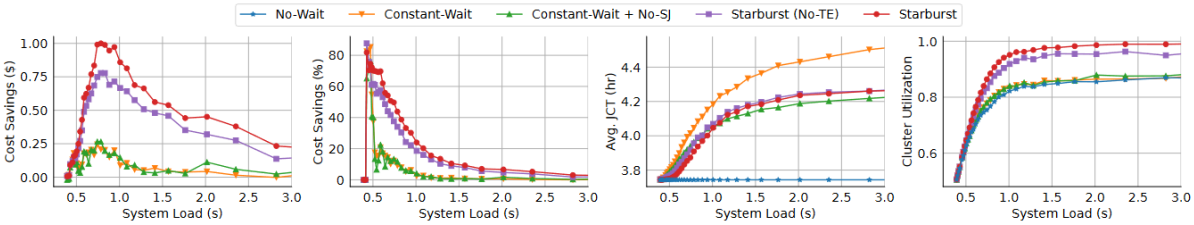
步骤

首先展示了真实集群上的结果和模拟器上的结果。

Load	Policy	Cost ($) Real / Sim	JCT (hr.) Real / Sim	Cluster Util. Real / Sim
s = 0.75	No Wait	23.18 / 25.66	0.70 / 0.74	66.65 / 66.80
	Constant Wait	21.33 / 21.33	0.72 / 0.77	70.39 / 71.95
	Starburst (No-TE)	4.96 / **5.15**	0.78 / 0.78	**72.59** / 70.77
	Starburst	**4.73** / 5.35	0.74 / 0.76	70.73 / **71.38**
s = 1.1	No Wait	45.97 / 48.76	0.62 / 0.66	83.90 / 83.50
	Constant Wait	33.89 / 37.06	0.67 / 0.72	83.30 / 84.25
	Starburst (No-TE)	9.79 / 10.53	0.71 / 0.77	84.7 / **87.78**
	Starburst	**8.96 / 7.13**	0.66 / 0.71	**85.20** / 87.69



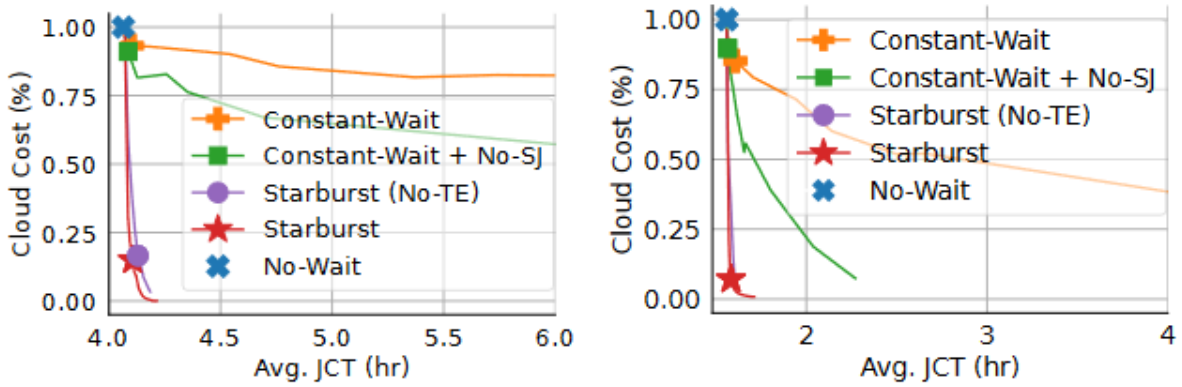
(a) End-to-end results over Helios (Venus).



(b) End-to-end results over Philly.

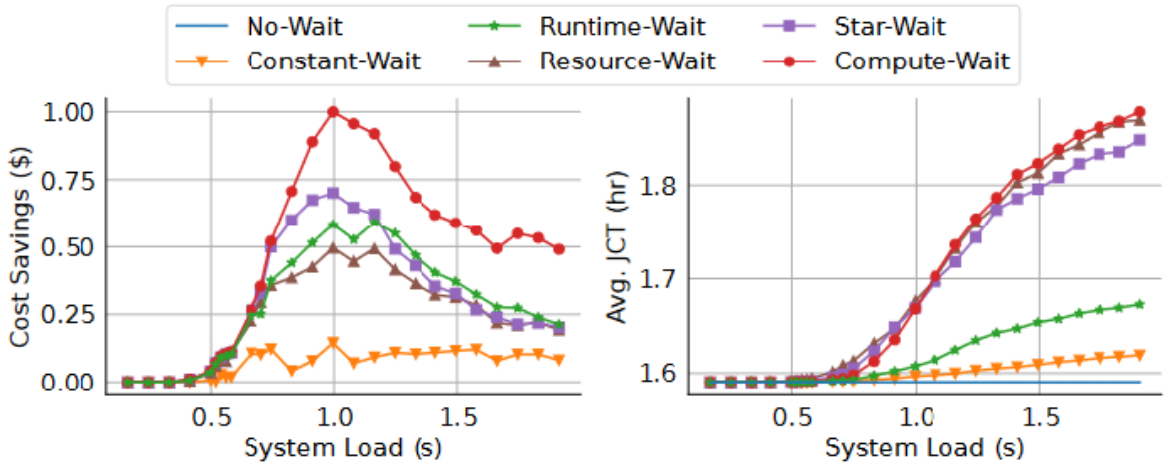
- Star-Wait即使没有时间估计，但可以达到和Compute-Wait类似的性能
- 模拟器对这些metrics是高度保真的

证明了star-wait也能达到和compute-wait这种有时间估计策略的类似的性能，starburst大幅拉低了帕累托曲线，并具备广泛的适用性。

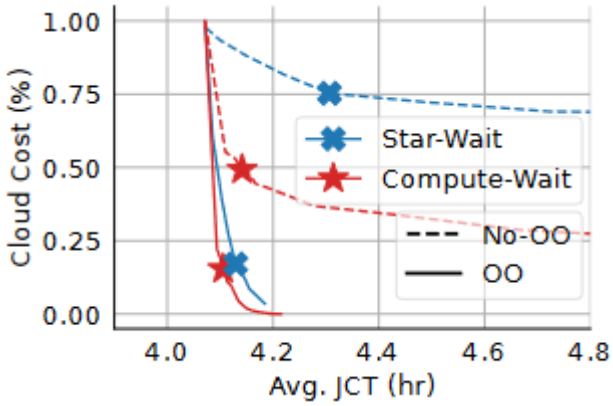


(a) Philly, $s = 0.83$. (b) Helios, $s = 0.85$

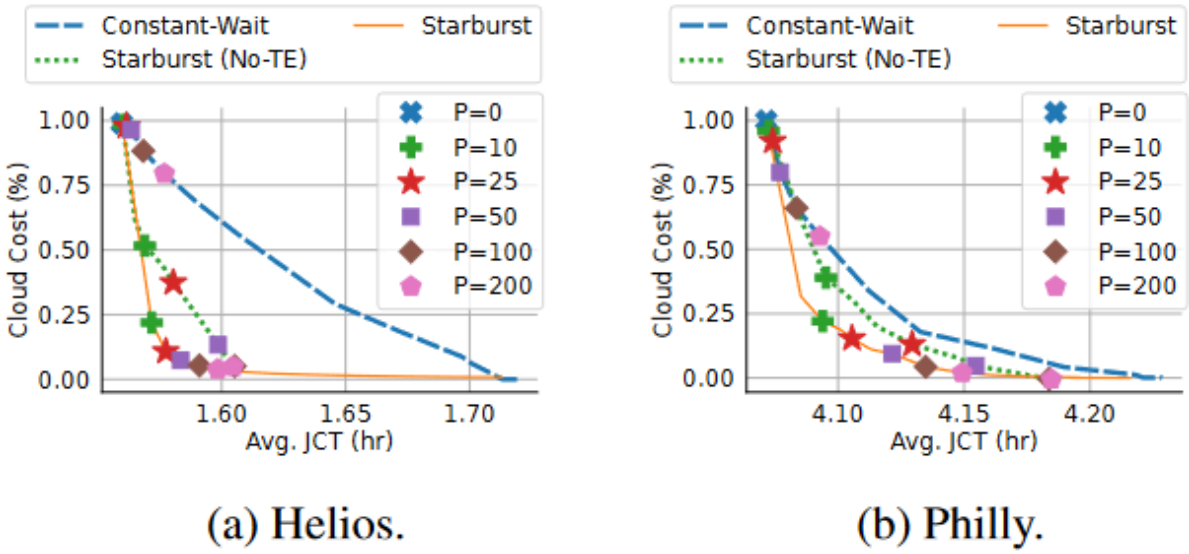
然后通过消融实验，分析了starburst的性能，包括等待策略：



OO调度：



等待预算B：



(a) Helios. (b) Philly.

也对比了starburst对长尾任务的等待时间和JCT的影响。

再和最优调度（MILP)做了对比（在一个500个作业的跨4节点的集群上），MILP假设了未来作业的全部知识，这是不现实的，所以可以帮助研究人员了解Starburst在现实条件下（即不知道未来作业到达情况）的表现与理论最优解的差距。

System Load	No-Wait	Constant-Wait	Starburst (No-TE)	Starburst	Optimal MILP	Avg. Waiting
s = 0.75	$443.40 (-0%)	$348.67 (-21.4%)	$201.54 (-54.5%)	$128.33 (**-71.1**%)	$121.89 (-72.5%)	0.03 hr.
s = 1	$760.68	$705.55 (-7.25%)	$607.31 (-20.1%)	$551.68 (**-27.5**%)	$526.80 (-30.7%)	0.05 hr.
s = 1.5	$1055.14	$912.86 (-13.5%)	$850.54 (-19.4%)	$817.35 (**-22.5**%)	$781.24 (-25.96%)	0.12 hr.

Starburst表现为在最优成本的5 %以内。

接着评估了starburst的鲁棒性：

- 消融DG(data gravity)实验，尽管starburst受到数据引力的强烈影响，但是与其他baselines相比，仍然表现出明显更好的成本和性能
- 在合成Philly的trace上使用Gamma分布的到达率，控制CV，评估对突发工作负载的鲁棒性。发现，CV越高starburst表现越好，因为它允许作业在突发阶段等待，并在干旱阶段计算执行。
- 通过不同的排队和装箱策略，证明了starburst对于底层的排队和装箱策略的不变性，改善等待策略是更为有效的。